Software Development Best Practices for Human-Rated Spacecraft

Project Management Challenge 2008
Daytona Beach, Florida
February 26 - 27, 2008

Michael Aguilar
NASA Engineering and Safety Center

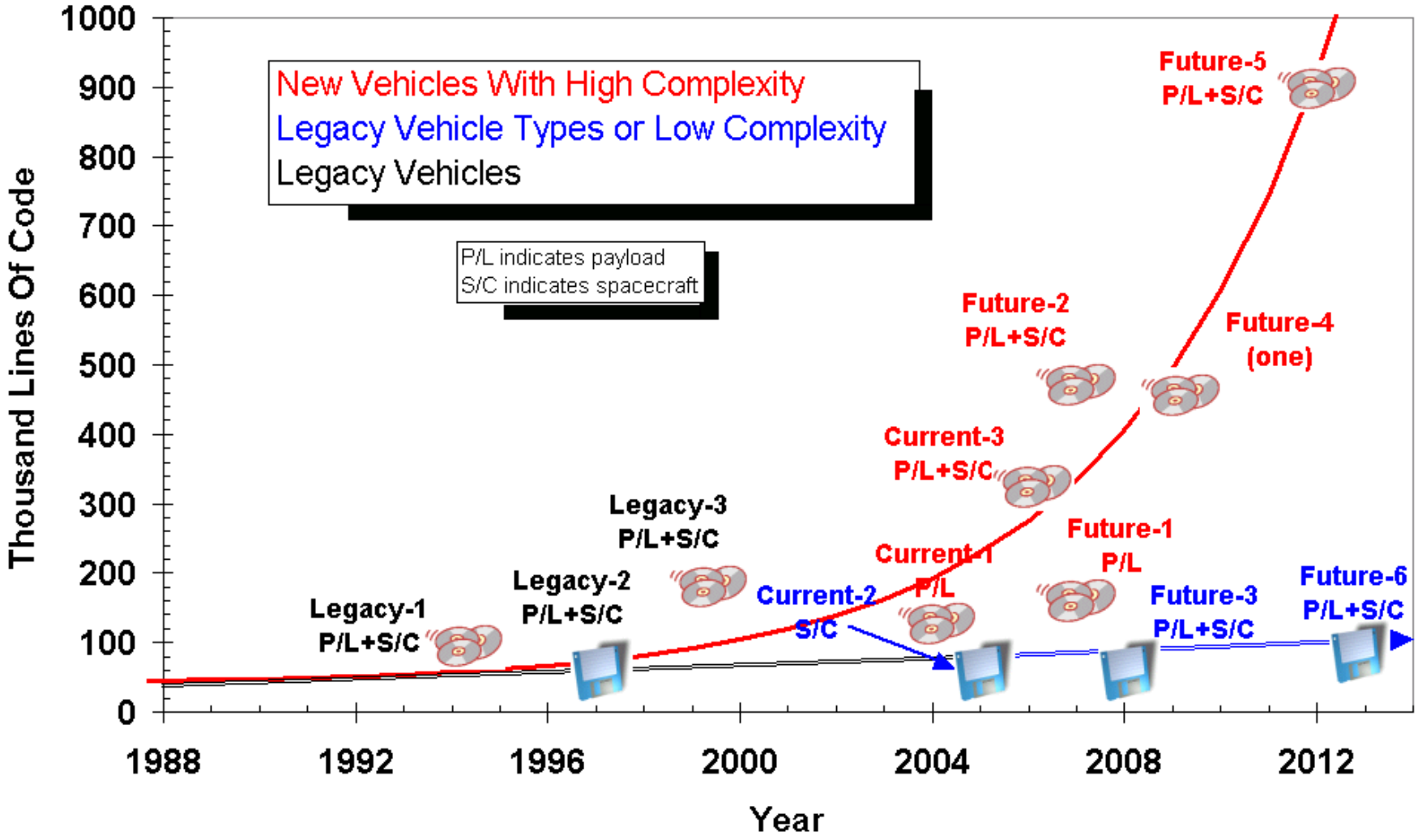
Background

- **In late 2005, the NASA Engineering and Safety Center (NESC) was asked by the Astronaut Office to answer the basic question:**

"How do you know you have a safe and reliable system?"

Full report available on-line at: nesc.nasa.gov

Trends In Space Vehicle Software Size



Software Reliability

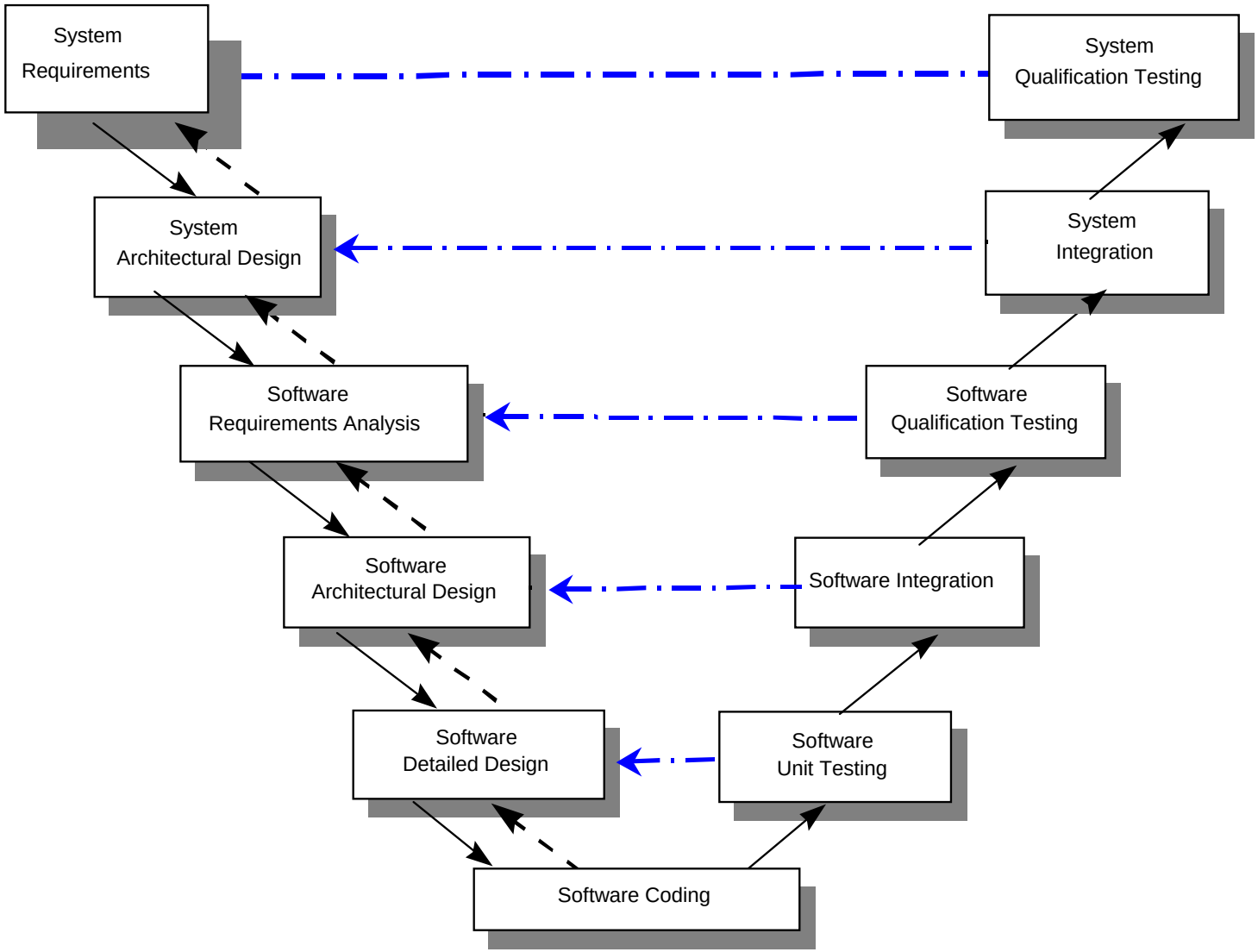
There are two major approaches to increasing the reliability of software:

- ***Software defect prevention (fault avoidance):*** using a disciplined development approach that minimizes the likelihood that defects will be introduced or will remain undetected in the software product
- ***Software fault tolerance:*** designing and implementing the software under an assumption that a limited number of residual defects will remain despite the best efforts to eliminate them

Software Defect Prevention

Using a disciplined development approach that minimizes the likelihood that defects will be introduced or will remain undetected in the software product.

Ideal Software Development Process



Allocation of Causes of Major Aerospace System Failures by Phase

Cause of failures	Phase				
	Reqts Def.	Arch. Def.	SW Design	Implementation	Test & Integration
Overconfidence and over reliance in digital automation	X	X	X		X
Not understanding the risks associated with software	X	X			X
Over relying on redundancy	X	X			
Confusing reliability and safety	X				X
Assuming that risk decreases over time	X				X
Ignoring early warning signs	X	X	X	X	X
Inadequate engineering	X	X	X		
Inadequate specifications	X				
Flawed review process	X	X	X	X	X
Inadequate safety engineering	X	X	X	X	X
Violation of basic safety engineering practices in the digital parts of the system	X	X	X	X	X
Software reuse without appropriate safety analysis	X	X	X	X	X
Unnecessary complexity and software functions	X		X	X	X
Operational personnel not understanding automation	X				X
Test and simulation environments that do not match the original environment	X				X
Deficiencies in safety-related information collection and use	X	X	X	X	X

Requirements Validation

- ***Modeling and simulation:*** Modeling and simulation should be used to set and evaluate performance parameters requirements affecting software performance, quality of communication services in the data transport layers, requirements for responses to failures and anomalous conditions, and human/software or system interactions.
- ***Non-advocate software and system requirements reviews*:** Reviews by knowledgeable third parties can uncover problems or issues that may have been overlooked by the primary requirements developers.
- ***Use of relevant "golden rules" and "lessons learned":*** Golden rules or lessons learned are excellent sources of requirements and should be reviewed as part of the requirements validation process.
- ***Hazards and Safety analyses:*** Hazards analyses, such as Failure Modes and Effects Analyses and Fault Trees.

Requirements Verification

- ***Early planning:*** Assure adequate planning for simulation test beds, communications equipment, test tools and data collection devices.
- ***Verification methods for low observable parameters:*** Instrumentation methods development for throughput, response time, or reliability**.**
- ***Anticipating ephemeral failure behaviors:*** The verification strategy should anticipate failure behaviors and plan for how this information can be captured – particular if they are ephemeral and non-reproducible.
- ***Testing of diagnostics and failure isolation capabilities:*** Diagnostic and fault isolation capabilities for off-nominal behavior.
- ***Capturing of unanticipated failures or behaviors:*** Ensure that test plans and procedures have provisions for recording of unanticipated behaviors.

Requirements Management

- ***Gathering resources for software test and verification:*** Assembling modeling and simulation capabilities, domain expertise, identifying areas of uncertainty
- ***Scheduling of software test and verification:*** The software test and verification program commonly includes managing critical path scheduling pressures.
- ***Tracking and maintaining requirements throughout the development process:*** Newly discovered software requirements should be propagated back into the higher level software and systems requirements documentation, and changes in existing requirements should be documented and tracked.
- ***Configuration Management of software requirements:*** Ensure that changes to software requirements are controlled and that when changes are made, they are propagated to all entities and stakeholders involved in the project.

Software Architecture Trades

- ***Distributed vs. centralized architectures:*** Distributed single point transmission failures such as undetected or unregulated message delays, or loss of synchronization in replicas of common data. Centralized architectures are vulnerable to failures in the central node of a centralized system.
- ***Extent of modularity:*** Uncoupled development, integration of revised components, and utilization of previously developed (or commercial off the shelf) components traded against increases the number of interfaces.
- ***Point-to-point vs. common communications infrastructure:*** The reduction of interdependencies among software elements and use of common inter-process communications constructs traded against vulnerabilities in terms of lost or delayed messages, message integrity, and message validity.

Software Architecture Trades

- ***COTS or reused vs. reused/modified vs. developed software:*** Reliability benefits are the ability to begin early test and integration of such software.
  - An understanding of the operational condition differences, constraints and tradeoffs are necessary. In safety critical applications, uncertainties about undocumented design decisions and tradeoffs embodied in the code may necessitate redevelopment.
    - Ariane 5 booster loss (inertial reference software re-use)
    - Magellan spacecraft (GNC software re-use from a DSCS satellite).
  - Verification of the suitability of the re-used software components by means of assessment of operational service history, the applicability of the allocated requirements to the published capabilities of the software, compatibility with other runtime elements, and proper version numbers.

Software Design

- ***Traceability:*** Requirements should be traceable to the functional elements or classes defined in the design.
- ***Exception handling and other failure behaviors:*** Exception handlers should consider all failure conditions defined in the requirements and in safety analyses. Where possible, exceptions should be handled as close to the locations in the code where they are generated.
- ***Diagnostics capabilities:*** Special attention should be paid to response time anomalies, priority inversion, and resource contention. The diagnostic capability of the system as a whole will largely depend on the diagnostic capabilities in all of the constituent software components.

Software Design

- ***Implementation language:*** The implementation language and runtime environment (including virtual machines) should be capable of realizing the design.
- ***Interfaces:*** Interfaces among software modules should be completely defined and include not only arguments for the inputs and outputs of the function or object itself but also additional parameters for status, error handling, and recovery. Interfaces should be designed “defensively”.
- ***Class library definition and inheritance:*** For object oriented architectures the definition of base and derived classes should be consistent and traceable to both the requirements and the architecture.

Software Design

- ***Compatibility with hardware and resource constraints:*** The software allocated to each hardware element should conform to memory, processor capacity, and interface constraints
- ***COTS and Non-developmental runtime elements:*** Existing software components and runtime elements should be configuration controlled, well characterized with respect to the intended use, and fully documented
- ***Automated Coding Tools:*** Newer techniques based on object oriented design or model-based development have resulted in tools that can go directly from design to executable code.
  - Among the advantages are the ability to generate an executable design that can be evaluated prior to detailed coding.
  - Among the concerns is the quality of the automatically generated code, particularly with respect to off-nominal conditions or inputs.

Code Implementation

- ***Use of "safe" subsets for safety or mission critical functions:*** Modern languages such as Ada, C, C++ and Java have safe subsets defined (or in the process of being defined) for their use in safety critical applications.
  - Disadvantages of such subsets is implementation in the language requires more source code which both reduces productivity (thereby adding to development cost), complicates software maintenance, and discourages reusability.
- ***Routine or class libraries, and runtime environments:*** The runtime libraries and other environmental components that support the developed software should conform to the constraints of the architecture and design and should provide the necessary capabilities to support desired failure behavior – including
  - Reliability, performance, throughput
  - Failure response, detection and recovery
  - Diagnostics requirements

Code Implementation

- ***Definition of suitable coding standards and conventions:*** Coding standards and conventions can enhance reliability by considering such issues as
  - Policies on dynamic memory allocation in safety critical systems (generally, it should not be allowed)
  - Policies on the use of “pointers”
  - “Defensive” coding practices for out of range inputs and response times
  - Exception handler implementation
  - Coding to enhance testability and readability
  - Documentation to support verification
  - Interrupt versus deterministic timing loop processing for safety critical software
  - Policies on allowable interprocess communications mechanisms (e.g., point to point vs. publish and subscribe)
  - Permitted use of dynamic binding (an alternative is static “case statements”)
  - Policies on initialization of variables (some standards prohibit assignment of dummy values to variables upon initialization in order to enable detection of assignment errors in subsequent execution)
  - Use of “friend” (C++) or “child” (Ada) declarations to enable testing and evaluation of encapsulated data code during development without requiring the subsequent removal of “scaffold code”.

Code Implementation

- ***Coding tools and development environments:*** Coding tools and integrated development environments can be used to for many purposes including automated documentation generation, enforcement of coding standards, debugging, diagnosis of potentially troublesome coding practices, cross reference listing, execution profiles, dependency analysis, design traceability, and many other purposes.
- ***Configuration management practices:*** Defect tracking and configuration management practices for software units and higher levels of integration should be defined to avoid uncertainty in the actual configuration of the software.

Test and Inspection of Code

- ***Code execution:*** Numerous methods for verification of executable code (see next slides), however code execution testing can not cover all possible code execution states
- ***Code inspections:*** Code inspections by knowledgeable individuals can find and fix mistakes overlooked in the initial programming. Another form of code inspection is the use of automated code analysis tools. Other types of reviews may occur in conjunction with code reviews including "walkthroughs" and "code audits"
- ***Formal methods:*** Testing is often insufficient to provide the necessary degree of assurance of correctness for safety critical software. Formal methods use mathematical techniques to prove the specification, the verification test suite and also automatic code generators to create the software. The NASA Langley Research Center has been active advancing formal methods, and extensive information is available from their web site.
- ***Cleanroom technique:*** The cleanroom technique was developed as an alternative approach to producing high quality software by preventing software defects by means of more formal notations and reviews prior to coding. The cleanroom technique has been used in several projects included the NASA/JPL Interferometer System Integrated Testbed.

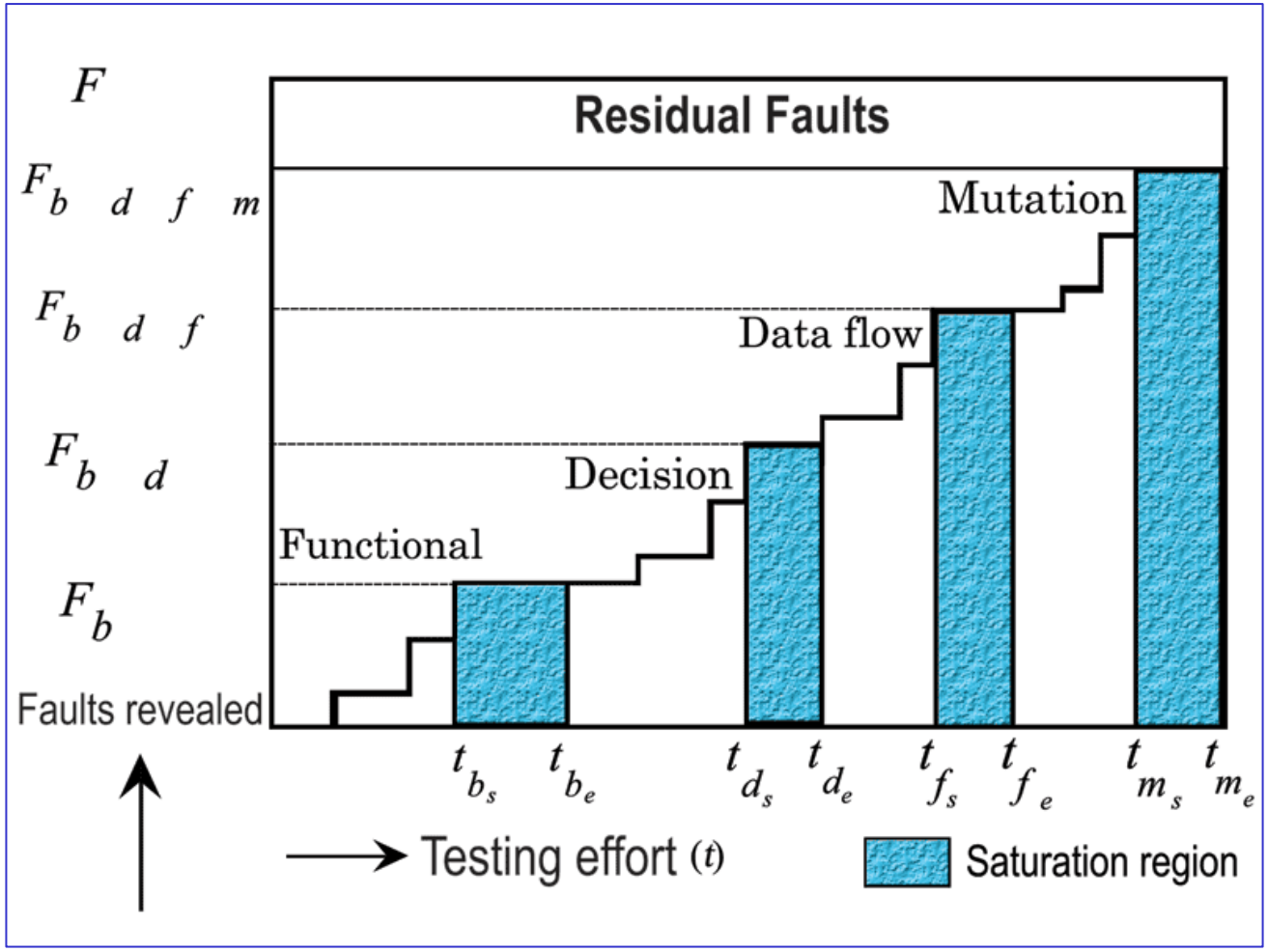
Types of Software Tests (1 of 2)

Method type and description	Objective	Test type	Applicable level
Scenario (also called thread) based testing: Testing using test data based on usage scenarios, e.g., simulation of the mission	Assess overall conformance and dependability in nominal usage	Black box	Integrated software and system
Requirements based testing: Testing to assess the conformance of the software with requirements	Determine whether the software meets specific requirements	Black box	All level at which requirements are defined
Nominal testing: Testing using input values within the expected range and of the correct type	Verify conformance with nominal requirements	Black box	All
Stress testing (a subcategory of negative testing): Testing with simulated levels of beyond normal workloads or starving the software of the computational resources needed for the workload; also called workload testing (usually run concurrently with endurance tests)	Measure capacity and throughput, evaluate system behavior under heavy loads and anomalous conditions, to determine workload levels at which system degrades or fails	Black box	Integrated software and system
Robustness testing (a subcategory of negative testing): Testing with values, data rates, operator inputs, and workloads outside expected ranges	Challenge or "break" the system with the objective of testing fail safe and recovery capabilities	Black & White box	All
Boundary value testing (a subcategory of negative testing): Test the software with data at and immediately outside of expected value ranges	Test error detection and exception handling behavior of software with anticipated exception conditions – whether software test item exits gracefully without an abnormal termination and for correctness	Black & White box	Unit, Software subsystem
Extreme value testing (a subcategory of negative testing): testing for large values, small values, and the value zero	Same as boundary value	Black & White box	Unit, Software subsystem

Types of Software Tests (2 of 2)

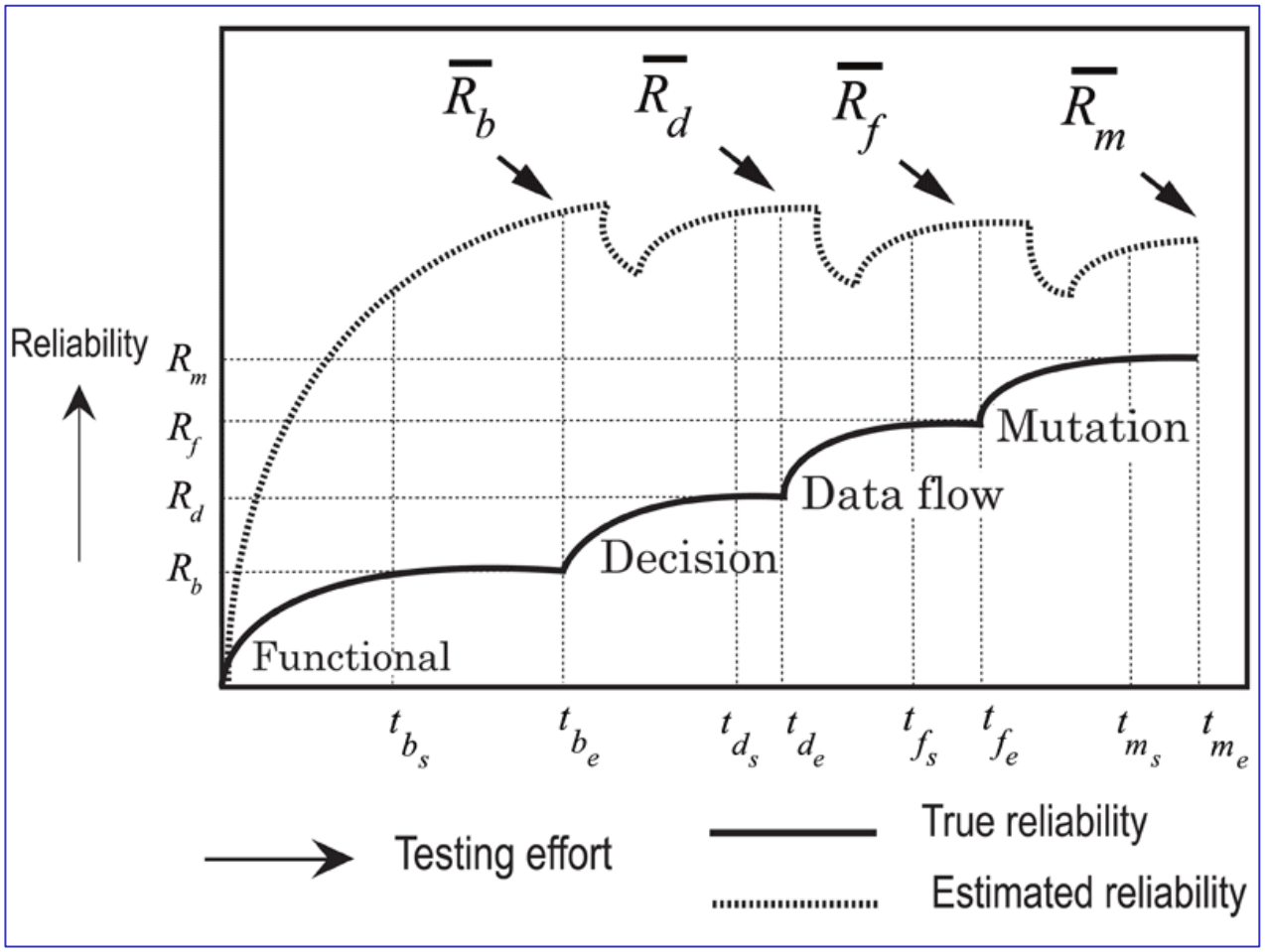
Method type and description	Objective	Test type	Applicable level
Random testing: test software using input data randomly selected from the operational profile probability distribution	Assess overall stability, reliability and conformance with requirements	Black box	Integrated system
Fault injection testing: Testing on the nominal baseline source code and randomly altered versions of the source (white box) or object code (black box)	Assess failure behavior, ensure that system properly responds to component failures	Black & White box	Integrated software
Branch testing: Test cases selected to test each branch at least once	Test correctness of code to the level of branches	White box	Software unit
Path testing: Test cases selected to test each path (i.e., feasible set of branches) at least once. Also called flow graph testing	Test correctness of code to the level of paths	White box	Software unit
Modified condition decision coverage (MCDC): Coverage–Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect that decision's outcome.	Test for safety critical software where a failure would probably or almost inevitably result in a loss of life	White box	Software unit (assembly code created by compiler under some circumstances)

Testing Saturation



Multiple testing methods increase detection of software faults

Reliability Estimation



Reliability based upon detected software failures can be misleading

Software Fault Tolerance

Designing and implementing the software under an assumption that a limited number of residual defects will remain despite the best efforts to eliminate them.

Replication

- **Replication: The executing redundant copies of software as an architecture level concept.**
  - **static redundancy: all copies of the executing program are provided with the same input and produce the same output. The output is chosen by a default selection of one of the channels or by comparing or voting on the output. The primary challenge for this form of replication is synchronization of the input or the output.**
  - **dynamic redundancy: one of the copies is assigned the active or "hot" role and other copies are in a standby role. The designation of "dynamic" is related to the fact that changing roles requires a change in the state of the software.**

Exception Handling

- **Exception handling: The ability of the software to detect (exception "raised" or "thrown") and handle (exception handled or "caught) abnormal conditions in a controlled manner which allows for continued operation or a safe shutdown and is an architectural or design-level concept.**
  - **Examples of responses include rollback and retry, substituting a default value, using a previous value, proceeding with processing without the input value, stopping processing, raising an alarm condition, sending a message to a user (or another software process) requesting an instruction on further action, or safe shutdown.**

Multiversion Software

- **Multiversion software: The independent generation of functionally equivalent programs, called versions, from the same initial specification. Independent generation of programs means that the programming efforts are carried out by different individuals or groups, using different programming languages and algorithms wherever possible. If three or more versions are developed, then voting logic can be implemented.**
  - **Even if the versions are developed independently, they suffer from correlated failures**
  - **Maintenance of multiple versions must be managed for all defect corrections and upgrades**

Recovery Blocks

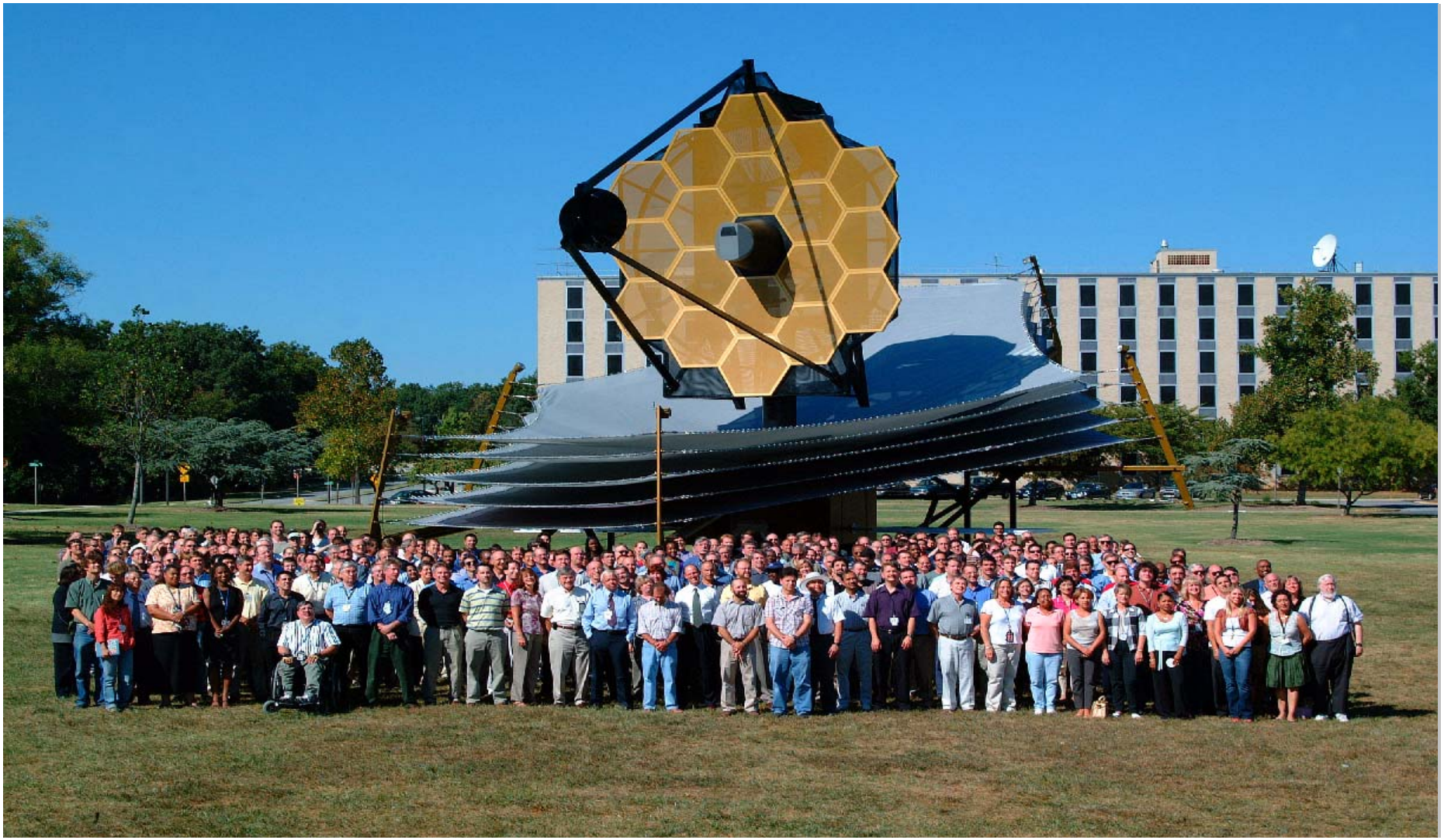
- **Recovery blocks: Structures that consist of three elements: a *primary routine*, an *alternate routine*, and a runtime *acceptance test*. If a primary routine is not capable of completing the task correctly, then an alternate routine can be invoked to mitigate the effects of the failure. The acceptance test determines whether the primary routine failed.**
  - **The primary and alternative routines can run sequentially or concurrently.**
  - **Recovery blocks are resource intensive not only in development, but also in subsequent maintenance and runtime resources.**

Computer Aided Software Engineering (CASE)

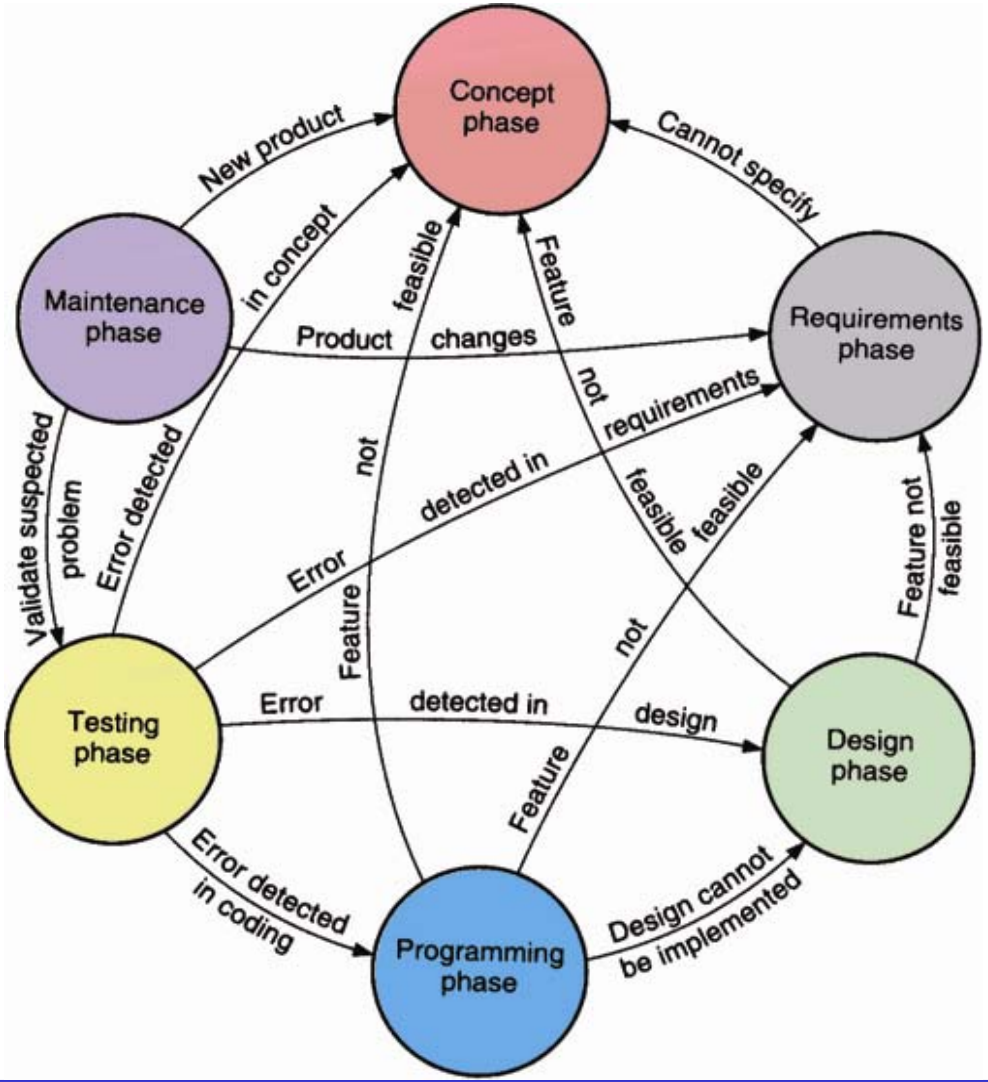
GSFC James Web Space Telescope

CASE experience

JWST Full-Scale Model



“Real” Software Development Process



JWST Software Development Toolset

- **Requisite Pro**
  - **Requirements Tracking**
  - **Requirements Tracing**
    - **Project Requirements**
    - **Test Cases**
    - **Personnel Assignments**
- **Clear Quest**
  - **Change Management and Tracking**
  - **Integrated with Requisite Pro and Clear Case**
- **Clear Case**
  - **Configuration Management**
- **Rational Rose Real Time**
  - **FSW Generation**

JWST Experience
Unified Modeling Language (UML)

- Scientists and Operators presented their requirements to JWST software developers in UML Use Case and Sequence Diagrams
- Review diagrams consistent within peer reviews
  - Review material consisted of minimal meeting package
  - Design was reviewed, projected directly out of the development environment
- Statechart Diagrams, Sequence Diagrams, and Timing Diagrams presented to Instrument developers to define software/hardware interface
- Design document described each and every software build exactly

JWST Software Integration Experience

- External software developers supplied with identical tool suite and hardware development environment
  - Rational ROSE tool suite, COTS C&DH hardware, Ground system, Database
- Complete C&DH model is supplied to NASA developers
- Library of C&DH model is supplied to ITAR restricted developers
- Integration of the C&DH model and instrument application specific model occurs for all development teams at the model level
- Training, lessons learned, support and guidance shared across all development teams due to identical environments and development tools
- Review presentation material was similar in content across all external and internal reviews

JWST Code Generation Experience

- Generated code is at interface between drivers, board-support package, OS, and state machine framework
  - Drivers, board-support package, OS configuration, and application specific functions are all hand coded
  - State machine diagrams provide structure to the reactive system
  - Source code for entire system is generated and available for analysis
- Minor and major alterations to system statemachine can be accomplished with little effort producing a robust statemachine after modification
- Minor defects and changes to functionality are accomplished with similar effort to hand coded implementation
- Static code analysis was performed on JWST source code (250K LOC) uncovering 60 defects
  - far below industry metrics for code defect density